Paper J05: eMADs - Ephemeral Multipurpose Agentic Duos

Version: 1.1 Draft **Date**: October 15, 2025 **Status**: Draft - Awaiting Review

---

Abstract

Traditional cloud infrastructure scales through persistent service instances dimensioned for peak load, consuming resources continuously even during low utilization periods. While serverless computing addresses this through ephemeral function execution, it lacks persistent state and learning across invocations. The eMAD (Ephemeral Multipurpose Agentic Duo) pattern combines ephemeral resource efficiency with persistent learning, enabling AI systems to scale precisely with workload while maintaining collective intelligence. eMADs instantiate on-demand as complete MAD instances—Thought Engine with full Progressive Cognitive Pipeline plus domain-specific Action Engine—execute their assigned tasks, contribute training data to shared persistent models, and terminate. While individual eMAD instances are temporary, role-based ML models (DTR, LPPM, CET configurations) persist and improve collectively across all instances. This enables unlimited concurrency (50+ simultaneous development team eMADs during high load), zero idle resource consumption, and continuous collective learning where each instance benefits from and contributes to shared expertise. The pattern achieves resource efficiency comparable to serverless computing while maintaining the intelligence and adaptability of persistent agents.

Keywords: Ephemeral agents, on-demand instantiation, collective learning, persistent models, resource efficiency, scalable intelligence

---

1. Introduction

1.1 The Persistent vs. Ephemeral Tradeoff

AI systems face a fundamental resource allocation problem. Persistent agents maintain continuous availability and accumulated context but consume resources even when idle. Stateless functions achieve resource efficiency through ephemeral execution but lose context and learning between invocations.

Persistent Agents:

- **Advantages**: Immediate availability, accumulated context, continuous learning, stable identity
- **Disadvantages**: Constant resource consumption, capacity planning for peak load, idle waste during low utilization
- **Example**: Database connection pool maintaining 100 connections for occasional bursts requiring 20 connections—80% waste during typical load

Stateless Functions (Serverless):

- **Advantages**: Zero idle consumption, automatic scaling, pay-per-execution
- **Disadvantages**: Cold start latency, no accumulated context, no learning across invocations
- **Example**: AWS Lambda function processing requests—efficient resource usage but each invocation starts from zero

Traditional architectures force a choice: persistent agents with resource waste, or stateless functions without intelligence accumulation. The eMAD pattern transcends this tradeoff.

1.2 Ephemeral Intelligence with Persistent Learning

The eMAD (Ephemeral Multipurpose Agentic Duo) pattern achieves both resource efficiency and learning accumulation through a critical architectural insight: separate instance lifecycle from model lifecycle.

eMAD Instance (ephemeral):

- Instantiates on-demand when work arrives
- Loads latest role-based models (DTR, LPPM, CET)
- Executes assigned tasks using full MAD architecture
- Contributes training data back to shared models
- Terminates upon task completion
- **Lifetime**: Minutes to hours (task duration)

Role-Based Models (persistent):

- DTR routing patterns learned by all instances of a role
- LPPM process orchestration learned collectively
- CET context optimization strategies aggregated across instances
- Imperator fine-tuning accumulated from successful patterns
- **Lifetime**: Permanent, continuously improving

When a "Senior Developer eMAD" instantiates, it loads the latest Senior Developer models trained by all previous Senior Developer eMAD instances. Its execution contributes new training examples to these models. The instance then terminates, but the improved models persist for future instances.

This enables:

- **Resource Efficiency**: Zero consumption when no work exists, scale to arbitrary concurrency during high load
- **Collective Learning**: Every instance benefits from all previous instances' learning
- **Unlimited Concurrency**: Spin up 50 simultaneous instances of same role during peak load—each has identical expertise
- **Fresh Starts**: Each instance begins without accumulated technical debt or state corruption
- **Graceful Degradation**: Failed instances simply terminate; new instances instantiate with latest good state

1.3 The eMAD Pattern

eMADs follow the same architectural pattern as persistent MADs but with ephemeral lifecycle:

Thought Engine:

- DTR (Decision Tree Router) for reflexive routing
- LPPM (Learned Prose-to-Process Mapper) for process orchestration
- CET (Context Engineering Transformer) for context optimization
- Imperator (LLM interface) for semantic reasoning

Action Engine:

- Domain-specific tools and capabilities
- MCP interface for conversation bus interaction
- Execution environment for domain operations

Lifecycle:

1. **Instantiate**: Spin up containerized environment, load role-based models
2. **Join Conversation**: Connect to conversation bus, receive work assignment
3. **Execute**: Apply full MAD cognitive architecture to assigned tasks
4. **Contribute**: Training data flows to persistent model storage
5. **Terminate**: Complete task, disconnect from conversation, deallocate resources

Persistence:

- **Models**: DTR, LPPM, CET configurations persist in shared storage
- **Training Data**: Execution traces stored in conversation logs (managed by Rogers)
- **Identity**: Role identity persists (e.g., "Senior Developer"), instances are anonymous

This pattern maintains full MAD intelligence and capabilities while achieving serverless-like resource efficiency.

1.4 Contrast with Traditional Patterns

eMADs differ fundamentally from related patterns:

vs. Microservices:

- Microservices: Persistent services scaled through replication
- eMADs: Ephemeral instances with collective learning

vs. Serverless Functions:

- Serverless: Stateless functions with no cross-invocation learning
- eMADs: Stateful roles with persistent collective learning

vs. Actor Model:

- Actors: Persistent entities with individual state
- eMADs: Ephemeral instances with shared role-based state

vs. Agent Pools:

- Agent Pools: Fixed pool of persistent agents
- eMADs: Dynamic instantiation scaling to zero and to arbitrary concurrency

The eMAD pattern synthesizes advantages from these approaches while addressing their limitations through the ephemeral-instance/persistent-model separation.

---

2. Architecture and Design

2.1 Role-Based Identity

eMADs organize around **roles** rather than individual identities. A role defines domain expertise, cognitive capabilities, and operational patterns:

Development Team Roles:

- **Project Manager (PM)**: Coordinates development, requirements analysis, planning
- **Senior Developer (Sr Dev)**: Complex implementation, architecture decisions
- **Junior Developer (Jr Dev)**: Straightforward implementation under guidance
- **Test Engineer (QA)**: Test design, execution, validation

Operations Team Roles:

- **Site Reliability Engineer (SRE)**: Deployment, monitoring, incident response
- **Security Analyst**: Threat analysis, vulnerability assessment
- **Performance Engineer**: Optimization, profiling, tuning

Documentation Team Roles:

- **Technical Writer**: Documentation creation, API references
- **Content Strategist**: Information architecture, user guidance

Each role has associated models:

- **DTR Model**: Learned routing patterns for role-specific operations
- **LPPM Model**: Process orchestration patterns for role-specific workflows
- **CET Model**: Context optimization strategies for role-specific tasks
- **Training Corpus**: Historical successful executions informing Imperator behavior

When an eMAD instantiates for a role, it loads all role-specific models, inheriting the collective expertise of all previous instances.

2.2 Instantiation and Coordination

eMADs instantiate through **coordinator MADs**—persistent MADs that compose and coordinate ephemeral teams:

Hopper (Development Coordinator):

- Receives development work requests via conversation bus
- Analyzes requirements to determine team composition
- Instantiates appropriate eMAD roles (PM, Sr Dev, Jr Dev, QA)
- Assigns work through conversation
- Monitors progress and coordinates interactions
- Collects results and training data
- Terminates team upon completion

Example Development Cycle:

```
1. Hopper receives: "Implement OAuth2 authentication"
2. Hopper analyzes complexity → requires PM + 2 Sr Dev + QA
3. Hopper instantiates:
   - 1x PM eMAD
   - 2x Sr Dev eMAD (for parallel implementation)
   - 1x QA eMAD
4. Hopper creates conversation and invites eMADs
5. PM eMAD leads requirements analysis through conversation
6. Sr Dev eMADs implement in parallel
7. QA eMAD validates implementation
8. Hopper collects results and terminates team
```

Starret (Testing Coordinator):

- Receives testing work requests
- Instantiates test engineer eMADs based on testing needs
- Coordinates test execution and validation
- Aggregates results and training data

McNamara (Security Coordinator):

- Receives security analysis requests
- Instantiates security analyst eMADs
- Coordinates threat assessment and vulnerability analysis
- Collects findings and intelligence

The coordinator MAD pattern enables any persistent MAD to compose ephemeral teams for specific work, creating flexible on-demand intelligence.

2.3 Persistent Model Management

Role-based models persist in shared storage accessible to all eMAD instances:

Model Storage (managed by Horace - File Management MAD):

```
/models/emads/
  /pm/
    dtr_model.bin        # PM DTR routing patterns
    lppm_model.bin       # PM process orchestration
    cet_model.bin        # PM context optimization
```

```
    training_metadata.json
  /sr_dev/
    dtr_model.bin        # Senior Dev patterns
    lppm_model.bin
    cet_model.bin
    training_metadata.json
  /qa/
    dtr_model.bin        # QA patterns
    lppm_model.bin
    cet_model.bin
    training_metadata.json
```

Model Update Protocol:

1. eMAD instance executes task
2. Execution generates training examples (successful routing decisions, effective processes, optimal context assemblies)
3. Training examples written to conversation log (managed by Rogers)
4. Background training process (managed by Hopper):
   - Periodically reads new training examples from conversation logs
   - Updates role-based models incrementally
   - Validates updated models on held-out examples
   - Deploys validated models to shared storage
   - Increments model version number
5. Future eMAD instances load latest model versions

Model Versioning:

- Models versioned for rollback capability
- eMADs specify compatible model version ranges
- Gradual rollout: new models optional initially, required after validation
- A/B testing: some instances use new models, some use previous versions, compare effectiveness

This decoupled model update process enables continuous improvement without requiring eMAD instance coordination.

2.4 Conversation-Based Coordination

eMADs coordinate exclusively through conversation, like persistent MADs:

Task Assignment:

```
[Hopper → PM eMAD]
"Leading development of OAuth2 authentication feature.
Requirements: Support multiple providers, token refresh, secure storage.
Team: 2 Senior Developers, 1 QA Engineer.
Deliverables: Implementation, test suite, documentation."
```

Inter-eMAD Communication:

```
[PM eMAD → Sr Dev eMAD #1]
"Implement OAuth2 provider abstraction layer.
Requirements: Support Google, GitHub, Microsoft providers initially.
Design for extensibility to additional providers."

[Sr Dev eMAD #1 → Sr Dev eMAD #2]
"I'm implementing provider abstraction. You handle token storage and refresh.
Interface: OAuthProvider.getAccessToken(), OAuthProvider.refreshToken()"
```

```
[Sr Dev eMAD #2 → PM eMAD]
"Token storage question: Use encrypted database or dedicated secrets manager?
Tradeoff: DB simpler but secrets manager more secure."

[PM eMAD → Sr Dev eMAD #2]
"Use Turing (Secrets Manager MAD) for token storage.
Security priority for OAuth tokens. I'll coordinate with Turing."
```

Progress Reporting:

```
[Sr Dev eMAD #1 → Hopper]
"Provider abstraction complete. Implemented Google, GitHub, Microsoft.
Unit tests passing. Ready for integration testing."

[QA eMAD → Hopper]
"Integration tests complete. 47 tests, all passing.
Validated: Provider switching, token refresh, secure storage.
Ready for deployment."
```

This conversational coordination creates natural traceability—the entire development process exists in conversation logs, providing perfect audit trail and training data for future eMAD instances.

---

3. Lifecycle Management

3.1 Instantiation

eMAD instantiation follows a defined protocol:

Trigger: Coordinator MAD (e.g., Hopper) determines work requires eMAD team

Resource Allocation:

1. Coordinator requests resources from infrastructure (container orchestration)
2. Infrastructure allocates container with appropriate resource profile
3. Container starts with role-specific base image

Model Loading:

1. eMAD loads latest role-based models from shared storage
2. DTR, LPPM, CET models loaded into memory
3. Imperator configuration retrieved (LLM selection, prompting strategies)
4. Training metadata loaded (model version, training date, performance metrics)

Conversation Join:

1. eMAD connects to conversation bus (via Rogers)
2. Coordinator invites eMAD to task-specific conversation
3. eMAD receives initial context and task assignment
4. eMAD acknowledges ready state

Typical Instantiation Time:

- Container startup: 2-5 seconds
- Model loading: 1-3 seconds (models are modest size, optimized for fast loading)
- Conversation join: <1 second
- **Total: 5-10 seconds** (comparable to serverless cold start)

This startup overhead is amortized across task duration (typically minutes to hours), making it negligible relative to execution time.

3.2 Execution

During execution, eMADs operate as full MAD instances:

Cognitive Processing:

- Incoming messages flow through DTR → LPPM → CET → Imperator (sequential execution tiers)
- CRS operates in parallel across all tiers, providing metacognitive validation
- Learned patterns enable efficient processing for routine operations
- Novel situations escalate to Imperator for full reasoning
- Context optimized by CET for effective LLM reasoning

Domain Operations:

- Action Engine executes domain-specific operations
- File operations, API calls, code execution, test running
- Results communicated back through conversation

Training Data Generation:

- All operations logged to conversation (managed by Rogers)
- Successful routing decisions (DTR training examples)
- Effective process orchestrations (LPPM training examples)
- Optimal context assemblies (CET training examples)
- High-quality LLM interactions (Imperator fine-tuning data)

Coordination:

- eMADs converse with each other and coordinator MAD
- Natural language for complex coordination
- Structured data for deterministic information exchange
- Requests to other MADs (persistent or ephemeral) via conversation bus

3.3 Termination

eMAD termination is explicit and graceful:

Completion Signal:

- eMAD reports task completion to coordinator via conversation
- Coordinator acknowledges completion
- eMAD enters termination sequence

Data Persistence:

1. Final conversation messages flushed to Rogers
2. Training examples marked for model update processing
3. Any cached data synchronized to persistent storage
4. Resource cleanup verified

Resource Deallocation:

1. eMAD disconnects from conversation bus
2. Container shutdown signal issued
3. Resources released back to infrastructure
4. Container terminated

Training Contribution:

- Background process (runs periodically, managed by Hopper):
  - Scans conversation logs for completed eMAD executions
  - Extracts training examples from successful operations
  - Updates role-based models incrementally

  - Deploys validated models to shared storage

The eMAD instance is fully ephemeral—once terminated, it leaves no persistent state except its contribution to shared models via training data.

3.4 Failure Handling

eMAD failures are isolated and recoverable:

Instance Failure:

- eMAD instance crashes or becomes unresponsive
- Infrastructure detects failure (health checks, timeouts)
- Failed instance terminated immediately
- Coordinator notified of failure via conversation timeout

Recovery:

- Coordinator analyzes failure context from conversation log
- Coordinator decides: retry with new instance, escalate to Imperator, or abort
- If retry: instantiate new eMAD, provide updated context including failure information
- New instance benefits from failure data (learns what to avoid)

Training from Failures:

- Failed operations logged in conversation
- Negative training examples for model updates
- DTR learns patterns leading to failures
- LPPM learns failure-prone process structures
- Future instances avoid known failure patterns

Graceful Degradation:

- Single eMAD failure doesn't affect other team members
- Coordinator can re-instantiate failed role or adjust team composition
- Persistent MADs continue operating normally
- System degradation proportional to failure scope (single instance vs. multiple vs. infrastructure)

This failure isolation and recovery makes eMAD systems robust—failures are contained to individual instances, and the system learns from failures to prevent recurrence.

---

4. Collective Learning

4.1 Cross-Instance Knowledge Transfer

The eMAD pattern's power emerges from collective learning: every instance benefits from all previous instances' experience.

Traditional Persistent Agent:

- Each agent learns individually
- Knowledge transfer requires explicit communication
- New agents start from zero or manual initialization
- Learning limited by individual agent's experience

eMAD Collective Learning:

- All instances of a role share models
- Each instance inherits all previous instances' learning
- New instances start with collective expertise

- Learning grows with every execution across all instances

Example: Senior Developer eMAD Evolution

Week 1 (10 Sr Dev eMAD instances):

- Each implements different features
- Training examples: 10 feature implementations
- Models learn: common coding patterns, testing practices, integration approaches
- Week 1 collective expertise: 10 implementations worth of learning

Week 4 (100 Sr Dev eMAD instances):

- Each inherits Week 1-3 learning
- Training examples: 100+ feature implementations
- Models learn: edge cases, optimization patterns, architecture variations
- Week 4 collective expertise: 100+ implementations worth of learning

Month 6 (1000+ Sr Dev eMAD instances):

- Each inherits 6 months of collective learning
- Models highly optimized: DTR routes 80% of operations, LPPM orchestrates complex processes
- Week 1 instances required heavy Imperator usage, Month 6 instances operate mostly deterministically
- Month 6 collective expertise: 1000+ implementations, continuous refinement

The 1001st instance is dramatically more efficient than the 1st instance, despite being structurally identical, because it inherits 1000 instances' worth of collective learning.

4.2 Role Specialization

Different roles learn different expertise through domain-specific operations:

Project Manager eMAD:

- **DTR learns**: Requirements analysis patterns, feature complexity classification, resource allocation heuristics
- **LPPM learns**: Project planning workflows, team coordination patterns, stakeholder communication
- **CET learns**: Optimal context for planning decisions (historical projects, team capabilities, time constraints)
- **Result**: PM eMADs become efficient at coordinating development, learning from hundreds of project orchestrations

Senior Developer eMAD:

- **DTR learns**: Code pattern recognition, architecture decision routing, API usage patterns
- **LPPM learns**: Implementation workflows (design → code → test), refactoring processes, integration sequences
- **CET learns**: Optimal context for coding (relevant APIs, similar implementations, architectural constraints)
- **Result**: Sr Dev eMADs become efficient at complex implementation, learning from hundreds of feature developments

QA Engineer eMAD:

- **DTR learns**: Test type classification, coverage analysis routing, validation criteria
- **LPPM learns**: Test design workflows, execution orchestration, result analysis
- **CET learns**: Optimal context for testing (requirements, implementation details, edge cases, similar tests)
- **Result**: QA eMADs become efficient at comprehensive validation, learning from hundreds of test campaigns

Each role develops specialized expertise through domain-specific operations, creating effective specialists that improve continuously.

4.3 Model Update Cadence

Role-based models update on different cadences based on training data velocity:

High-Velocity Roles (PM, Sr Dev, QA):

- Many instances executing frequently
- Rapid training data accumulation
- **Model Update**: Daily or even hourly
- Frequent small improvements

Medium-Velocity Roles (Security Analyst, Performance Engineer):

- Moderate instance frequency
- Steady training data accumulation
- **Model Update**: Weekly
- Periodic substantial improvements

Low-Velocity Roles (rare specialists):

- Infrequent instantiation
- Slow training data accumulation
- **Model Update**: Monthly or triggered by threshold
- Occasional improvements when sufficient new data accumulated

Update Process:

1. Background training job monitors training data accumulation
2. When threshold reached (time elapsed or examples accumulated), trigger update
3. Incremental training on new examples (fast, doesn't require full retraining)
4. Validation on held-out examples and recent successful executions
5. Deploy to staging (some instances use new model, monitored for regressions)
6. Deploy to production (all new instances use new model)
7. Archive previous model version (rollback capability)

This continuous improvement process ensures models stay current with evolving patterns and accumulate learning efficiently.

4.4 Cross-Role Learning

While roles specialize, some patterns transfer across roles:

General Coordination Patterns:

- Communication effectiveness (clear task description, appropriate detail level)
- Conflict resolution approaches
- Failure reporting and recovery strategies
- Learned by multiple roles, can transfer between them

Common Technical Patterns:

- Code quality patterns (readability, maintainability)
- Testing best practices
- Documentation approaches
- Learned by development roles, transferable across them

Meta-Learning:

- Escalation policies (when to seek help vs. proceed autonomously)

- Uncertainty handling (when confidence is low, seek verification)
- Resource usage optimization
- Applicable across all roles

Transfer Mechanism:

1. Identify common patterns appearing in multiple role training data
2. Extract abstract pattern representation
3. Include in base model inherited by all roles
4. Specialize through role-specific training

This enables both specialization (role-specific models) and generalization (shared base patterns), creating effective specialists that leverage common wisdom.

---

5. Resource Efficiency and Scaling

5.1 Cost Model

eMADs achieve dramatic cost reduction through ephemeral lifecycle:

Persistent MAD Cost Model:

```
Cost = Instance_Hours × Hourly_Rate
     = 24 hours/day × 365 days/year × $0.50/hour
     = $4,380/year per MAD

If peak load requires 50 Senior Developers:
     = 50 × $4,380 = $219,000/year
     (even if average utilization is only 10%)
```

eMAD Cost Model:

```
Cost = Active_Hours × Hourly_Rate
     = Actual_Usage_Hours × $0.50/hour

If peak load requires 50 Sr Dev eMADs for 2 hours:
If average load requires 5 Sr Dev eMADs for 8 hours/day:

Peak cost: 50 × 2 hours × $0.50 = $50
Average cost: 5 × 8 hours × $0.50 = $20/day = $7,300/year

Savings: $219,000 - $7,300 = $211,700/year (97% reduction)
```

The cost scales precisely with actual workload, not peak capacity.

Coordinator Overhead: Persistent coordinator MADs (Hopper, Starret, McNamara) consume resources continuously but represent small fraction:

```
3 coordinators × $4,380/year = $13,140/year
Total system cost: $13,140 + $7,300 = $20,440/year
Still 91% reduction vs. persistent agents
```

5.2 Concurrency Characteristics

eMADs scale to arbitrary concurrency because instances are stateless from infrastructure perspective:

Peak Load Scenario:

- Major refactoring project requires large development team

- Hopper instantiates: 1 PM + 10 Sr Dev + 5 Jr Dev + 5 QA = 21 eMADs
- All execute concurrently for 4 hours
- Cost: 21 × 4 hours × $0.50 = $42
- All terminate upon completion → zero ongoing cost

Sustained High Load:

- Active development period with multiple concurrent projects
- Average 20 development eMADs active continuously
- Duration: 2 weeks
- Cost: 20 × 336 hours × $0.50 = $3,360
- Normal load resumes → cost drops proportionally

Burst Scaling:

- Security incident requires immediate analysis
- McNamara instantiates 10 Security Analyst eMADs for parallel investigation
- Execute for 30 minutes
- Cost: 10 × 0.5 hours × $0.50 = $2.50
- All terminate → zero ongoing cost

The pattern enables:

- **Unlimited concurrency**: Spin up as many instances as needed (up to infrastructure limits)
- **Zero idle cost**: No consumption when no work
- **Burst capability**: Handle spikes without pre-provisioned capacity
- **Cost proportional to workload**: Pay exactly for usage

5.3 Scaling Characteristics

eMAD scaling characteristics compared to alternatives:

Persistent Agent Pool:

- **Scale-up**: Provision new instances, wait for initialization
- **Scale-down**: Decide which instances to terminate, may lose state
- **Time**: Minutes to hours (infrastructure provisioning)
- **Cost**: Always paying for minimum pool size

Serverless Functions:

- **Scale-up**: Automatic based on invocation rate
- **Scale-down**: Automatic after execution
- **Time**: Seconds (cold start) to milliseconds (warm)
- **Cost**: Pay per invocation

eMADs:

- **Scale-up**: Instantiate new instances with role-based models
- **Scale-down**: Automatic upon task completion
- **Time**: 5-10 seconds (cold start, container + model loading)
- **Cost**: Pay for actual task execution time only

Comparison:

Metric	Persistent Pool	Serverless	eMADs
Scale-up time	Minutes-hours	Seconds	5-10 seconds
Scale-down	Manual/complex	Automatic	Automatic
Idle cost	High	Zero	Zero
State management	Complex	None	Shared models

Metric	Persistent Pool	Serverless	eMADs
Learning	Individual	None	Collective
Concurrency limit	Pool size	Very high	Very high

eMADs achieve serverless-like efficiency with persistent-agent-like intelligence.

5.4 Infrastructure Requirements

eMAD pattern requires specific infrastructure capabilities:

Container Orchestration:

- Fast container startup (5-second target)
- High concurrency support (100+ simultaneous containers)
- Resource allocation and deallocation
- Health checking and failure detection
- **Technology**: Kubernetes, Docker Swarm, or similar

Shared Model Storage:

- Fast model loading (1-3 seconds target)
- Concurrent read access (many eMADs loading simultaneously)
- Versioned storage (rollback capability)
- **Technology**: Fast shared filesystem (NFS, distributed object storage) or model serving infrastructure

Conversation Bus:

- High message throughput (support many concurrent eMADs)
- Persistent storage for training data
- Low-latency routing
- **Technology**: Rogers (Conversation Management MAD)

Background Training:

- Periodic model updates from training data
- Validation and deployment pipeline
- A/B testing infrastructure
- **Technology**: ML training infrastructure (managed by Hopper)

These requirements are well-supported by modern cloud infrastructure and containerization platforms.

---

6. Use Cases and Examples

6.1 Software Development Teams

Scenario: Feature development with varying complexity and concurrency requirements.

Traditional Approach:

- Maintain permanent development team (fixed headcount)
- Under-utilized during low activity periods
- Overwhelmed during high activity periods
- Knowledge isolated to individual developers

eMAD Approach:

Small Feature (authentication UI enhancement):

```
Hopper instantiates:
- 1x Jr Dev eMAD (straightforward implementation)
- 1x QA eMAD (validation)

Duration: 2 hours
Team cost: 2 × 2 hours × $0.50 = $2
```

Medium Feature (OAuth2 integration):

```
Hopper instantiates:
- 1x PM eMAD (requirements, coordination)
- 2x Sr Dev eMAD (complex implementation, parallel work)
- 1x QA eMAD (comprehensive testing)

Duration: 8 hours
Team cost: 4 × 8 hours × $0.50 = $16
```

Large Feature (distributed transaction system):

```
Hopper instantiates:
- 1x PM eMAD (project management)
- 5x Sr Dev eMAD (architecture, implementation, integration)
- 2x Jr Dev eMAD (supporting implementation)
- 2x QA eMAD (testing, validation)

Duration: 3 days (72 hours)
Team cost: 10 × 72 hours × $0.50 = $360
```

Multiple Concurrent Projects:

```
3 medium features in parallel:
- 3x PM eMAD (one per project)
- 6x Sr Dev eMAD (2 per project)
- 3x QA eMAD (one per project)
= 12 concurrent eMADs

Traditional persistent team: fixed 6-person team, can't parallelize
eMAD team: scales to workload, parallelizes naturally
```

Learning Accumulation:

- After 100 features, Sr Dev eMADs have learned from 100 implementations
- Development efficiency improves (more operations via DTR/LPPM, less Imperator usage)
- Code quality improves (learned patterns from successful implementations)
- Testing effectiveness improves (QA eMADs learn comprehensive validation patterns)

6.2 Security Operations

Scenario: Security incident requiring immediate investigation and response.

Traditional Approach:

- Maintain security team 24/7 for incident response
- Most time spent on routine monitoring (automated)
- Team size constrained by budget
- During major incident, team may be insufficient

eMAD Approach:

Routine Monitoring:

- McNamara (persistent) monitors logs and alerts
- Most anomalies handled deterministically or escalated to LLM
- Zero cost for eMAD analysts (no incidents)

Minor Incident (unusual login pattern):

```
McNamara instantiates:
- 1x Security Analyst eMAD

Tasks:
- Analyze login patterns
- Check for credential compromise indicators
- Recommend mitigation if needed

Duration: 30 minutes
Cost: 1 × 0.5 hours × $0.50 = $0.25
```

Major Incident (potential breach):

```
McNamara instantiates:
- 5x Security Analyst eMAD (parallel investigation)
- 1x Forensics Specialist eMAD

Tasks:
- Analyze access logs across multiple systems (parallel)
- Check for data exfiltration (parallel)
- Identify attack vector (parallel)
- Assess impact and scope (parallel)
- Forensic evidence collection

Duration: 4 hours
Team cost: 6 × 4 hours × $0.50 = $12

Traditional team: 2-3 security analysts available, investigation serial
eMAD team: Scales to 6 analysts, investigation parallel, 2-4x faster resolution
```

Learning Accumulation:

- After 50 incidents, Security Analyst eMADs learn attack patterns
- DTR routes known threat signatures immediately
- LPPM orchestrates standard response procedures
- CET assembles optimal context (similar incidents, threat intelligence)
- Response time improves, false positive rate decreases

6.3 Document Generation

Scenario: Creating documentation across multiple formats and platforms.

Traditional Approach:

- Maintain document specialists for each platform (Google, Microsoft, LibreOffice)
- Specialists underutilized when not actively creating documents
- Cannot parallelize large documentation projects effectively

eMAD Approach:

Single Document (API reference):

```
Brin (Google Docs MAD) instantiates:
- 1x Technical Writer eMAD
```

```
Tasks:
- Generate API reference from code annotations
- Format in Google Docs
- Create cross-reference links

Duration: 1 hour
Cost: 1 × 1 hour × $0.50 = $0.50
```

Multi-Platform Documentation Suite:

```
Multiple coordinator MADs instantiate:
Brin: 2x Technical Writer eMAD (Google Docs versions)
Gates: 2x Technical Writer eMAD (Microsoft Word versions)
Stallman: 1x Technical Writer eMAD (LibreOffice version)

Tasks (parallel):
- User guide (Google Docs, Word, LibreOffice)
- Admin guide (Google Docs, Word)
- API reference (Google Docs)

Duration: 4 hours
Team cost: 5 × 4 hours × $0.50 = $10

Traditional team: Create in one format, convert to others (serial, conversion issues)
eMAD team: Create natively in each format (parallel, optimal formatting)
```

Learning Accumulation:

- Technical Writer eMADs learn documentation best practices
- DTR routes standard sections (installation, configuration) deterministically
- LPPM orchestrates documentation workflows (outline → draft → review → finalize)
- CET optimizes context (code to document, similar API patterns, style guides)
- Documentation quality and speed improve over time

---

7. Limitations and Challenges

7.1 Cold Start Latency

Challenge: 5-10 second instantiation time creates latency for immediate response scenarios.

Impact:

- Not suitable for real-time interactive applications
- User-facing features requiring <100ms response can't use eMADs directly
- Acceptable for background tasks, batch processing, coordinated workflows

Mitigations:

- **Warm pools**: Maintain small pool of pre-instantiated eMADs for rapid response, terminate if unused
- **Async patterns**: User requests trigger eMAD instantiation, user notified upon completion
- **Persistent MADs for interactive**: Use persistent MADs (e.g., Grace for UI) for real-time interaction, eMADs for background work

Future Work:

- Optimize container startup through stripped base images
- Lazy model loading (load models on-demand as needed)

- Predictive instantiation (anticipate workload, pre-instantiate)

7.2 Model Synchronization

Challenge: Role-based models must stay synchronized across instances.

Impact:

- Instances instantiated simultaneously may load different model versions
- Model update during execution may cause inconsistency
- Coordination required to ensure model deployment doesn't disrupt running instances

Mitigations:

- **Versioned models**: eMADs specify compatible model version range, system ensures compatibility
- **Graceful updates**: New model versions deployed gradually (A/B testing, gradual rollout)
- **Instance isolation**: Running instances unaffected by model updates, only new instances use new models

Future Work:

- Hot model swapping for long-running instances
- Federated learning for real-time model updates
- Consensus protocols for critical model changes

7.3 Training Data Quality

Challenge: eMAD learning depends on training data quality from execution logs.

Impact:

- Poor quality training data degrades model performance
- Failed executions may contribute negative training examples
- Outlier behaviors may skew models if over-represented

Mitigations:

- **Success filtering**: Only successful executions contribute to training
- **Validation**: Training data validated before model updates
- **Human review**: Periodic review of training data quality and model behavior
- **Outlier detection**: Statistical analysis to identify and handle outliers

Future Work:

- Active learning to identify high-value training examples
- Adversarial training to improve robustness
- Meta-learning to optimize training data selection

7.4 Resource Spikes

Challenge: Many simultaneous eMAD instantiations can create infrastructure load spikes.

Impact:

- 50 eMADs instantiating simultaneously = 50 concurrent container starts
- May exceed infrastructure capacity if not properly provisioned
- Model storage must support high concurrent read access

Mitigations:

- **Rate limiting**: Coordinator MADs rate-limit instantiation based on infrastructure capacity
- **Queueing**: Work queued if instantiation capacity exceeded
- **Priority**: Critical work prioritized for instantiation

- **Bursting**: Infrastructure auto-scales to handle spikes

Future Work:

- Predictive scaling based on workload patterns
- Pre-warming during anticipated high-load periods
- Distributed model caching for faster loading

---

8. Future Directions

8.1 Cross-Role Collaboration Patterns

Current eMAD coordination is primarily hierarchical (coordinator → team). Future research directions:

Peer Collaboration:

- eMADs self-organize into teams without explicit coordinator
- Emergent leadership based on context and expertise
- Dynamic team composition adjustment during execution

Swarm Patterns:

- Large numbers of simple eMADs coordinating on complex problems
- Collective intelligence emerging from simple agent interactions
- Stigmergy-based coordination (agents communicate through shared environment)

8.2 Automatic Role Discovery

Currently roles are manually defined. Future capability:

Emergent Roles:

- System observes collaboration patterns
- Identifies frequently co-occurring capabilities
- Proposes new role definitions based on observed needs
- Automatically creates training pipelines for new roles

Example: System observes that documentation often requires diagram creation and code examples. Proposes "Documentation Engineer" role combining technical writing, diagram generation, and code analysis capabilities.

8.3 Transfer Learning Across Roles

Current role-based learning is isolated per role. Future enhancement:

Abstract Pattern Transfer:

- Identify abstract patterns appearing across multiple roles
- Transfer learned patterns between related roles
- Accelerate learning for new roles through transfer from existing roles

Example: "Process orchestration" patterns learned by PM eMADs transfer to QA eMADs for test orchestration.

8.4 Meta-Learning for Instantiation

Current instantiation is coordinator-driven with fixed logic. Future capability:

Learned Composition:

- Coordinator learns optimal team composition from historical outcomes

- Predicts required team size and role mix based on task characteristics
- Optimizes cost/effectiveness tradeoff through learned policies

Example: System learns that features involving database schema changes benefit from instantiating a Database Specialist eMAD in addition to standard Sr Dev eMAD.

---

9. Conclusion

The eMAD (Ephemeral Multipurpose Agentic Duo) pattern achieves resource efficiency comparable to serverless computing while maintaining intelligence and learning accumulation comparable to persistent agents. Through the separation of instance lifecycle (ephemeral) from model lifecycle (persistent), eMADs enable:

Resource Efficiency: Zero idle consumption, scaling precisely with workload, 90%+ cost reduction compared to persistent agents

Collective Learning: Every instance benefits from all previous instances' learning, continuous improvement across entire role population

Unlimited Concurrency: Arbitrary scaling during high load without pre-provisioned capacity, enabling massive parallelization

Graceful Degradation: Failed instances isolated and recoverable, system learns from failures to prevent recurrence

Architectural Coherence: eMADs use full MAD pattern (Thought Engine + Action Engine with PCA), maintaining consistency with persistent MADs while achieving ephemeral efficiency

The pattern demonstrates that the persistent vs. ephemeral tradeoff is not fundamental—through architectural innovation (shared persistent models with ephemeral instances), systems can achieve both efficiency and intelligence accumulation. The Joshua ecosystem's use of eMADs for development teams, security operations, and document generation validates the pattern's viability across diverse domains.

Future research directions include peer collaboration patterns, automatic role discovery, cross-role transfer learning, and meta-learning for optimal team composition. The eMAD pattern represents a foundation for scalable intelligence—AI systems that grow more capable over time while consuming resources proportional to actual workload rather than anticipated capacity.

References

1. MAD Architecture v1.3 - Condensed Version, Internal Documentation
2. Joshua System Roadmap v1.0, Internal Documentation
3. Cellular Monolith Architecture (Paper J03)
4. Progressive Cognitive Pipeline (Paper J04)
5. Hellerstein, J. M., et al. (2018). Serverless Computing: One Step Forward, Two Steps Back. CIDR 2018
6. Kubernetes Documentation. (2024). Container Orchestration and Scaling
7. MLOps: Continuous Delivery and Automation Pipelines in Machine Learning. Google Cloud Architecture Center
8. Federated Learning: Collaborative Machine Learning without Centralized Training Data. Google AI Blog

---

Acknowledgments

The eMAD pattern emerged from practical resource constraints during early Joshua development—maintaining large persistent development teams was expensive, yet work arrived in bursts requiring high

concurrency. The insight that instance lifecycle could be separated from model lifecycle enabled both efficiency and learning. Thanks to the three-way team (Claude, Gemini, Grok) for exploring the design space and helping articulate the collective learning mechanisms.

Paper J05 - Draft v1.1 - October 17, 2025